Exercice 1. Tri par sélection

Question 1 –

```
def mini(L):
    assert L != []
    i_min = 0
    for i in range(1,len(L)):
        if L[i] < L[i_min]:
            i_min = i
    return i_min
```

```
def supprimer(L, i):
    assert 0 <= i and i < len(L)
    return L[:i] + L[i+1:]
```

```
def tri_selection(L):
    T = []
    for _ in range(len(L)):
        i = mini(L)
        T.append(L[i])
        L = supprimer(L, i)
    return T
```

Question 2 – Le temps d'exécution des fonctions `mini` et `supprimer` est en $\Theta(\texttt{len(L)})$. Donc la complexité de la fonction `tri_selection` est en $\Theta(\texttt{len(L)}^2)$.

Question 3 – Ce n'est pas un tri en place car une nouvelle liste est créée.

C'est bien un tri stable car dans la fonction `mini`, si le plus petit élément apparaît plusieurs fois, alors on renvoie le plus petit indice possible. Ainsi, le tri ne serait pas stable si on remplaçait la condition « `L[i] < L[i_min]` » par « `L[i] <= L[i_min]` ».

Exercice 2. Tri par insertion

Question 1 –

```
# Hypothèse: T est triée
def insertion(T, x):
    i = 0
    while i < len(T) and T[i] < x:
        i += 1
    return T[:i] + [x] + T[i:]
```

```
def tri_insertion(L):
    T = []
    for e in L:
        T = insertion(T, e)
    return T
```

Question 2 – La fonction `insertion` est en complexité $\Theta(\texttt{len(T)})$. La fonction `tri_insertion` est en complexité $\Theta(\texttt{len(L)}^2)$.

Question 3 – Ce n'est pas un tri en place car une nouvelle liste est créée.

Ce n'est pas un tri stable, par exemple pour `[1, 1.0]` on obtient `[1.0, 1]`. En effet dans la boucle `for` de la fonction `insertion`, si `x == T[i]` alors on renvoie `L + [x] + T[i:]` alors que `T[i]` est placé avant `x` dans `L`. Pour obtenir un tri stable, il faut changer la condition « `T[i] < x` » par « `T[i] <= x` ».

Question 4 – La recherche dichotomique dans une liste triée `T` est en complexité $\mathcal{O}$`(log(len(T)))`. Après avoir obtenu l'indice d'insertion, il faut insérer l'élément dans `T`, ce qui nécessite de faire une copie de la liste, soit une complexité en Θ`(len(T))`. Finalement, même avec une recherche dichotomique, le temps d'exécution est en Θ`(len(L)`^{2}`)`.

Exercice 3. Tri rapide

Question 1 –

```
# L[0] est le pivot
# Hypothèse: L est non vide
def separer(L):
    L1 = []; L2 = []
    for i in range(1,len(L)):
        if L[i] <= L[0]:
            L1.append(L[i])
        else:
            L2.append(L[i])
    return L1, L2
```

```
def tri_rapide(L):
    if L == []:
        return []
    else:
        L1, L2 = separer(L)
        T1 = tri_rapide(L1)
        T2 = tri_rapide(L2)
        return T1 + [L[0]] + T2
```

Question 2 – Dans le pire cas, à chaque appel récursif l'une des deux sous-listes est vide. Dans ce cas, le nombre d'appels récursifs est en Θ`(len(L))`. Comme la fonction `separer` est en complexité Θ`(len(L))`, la fonction `tri_rapide` est en complexité Θ`(len(L)`^{2}`)` dans le pire cas.

Question 3 – Ce n'est pas un tri en place car une nouvelle liste est créée.

Ce n'est pas un tri stable, par exemple pour `[1, 1.0]` on obtient `[1.0, 1]`. En effet, les éléments égaux au pivot (qui est `L[0]`) sont stockés dans la liste `L1` et apparaissent donc avant `L[0]` dans `T`. Pour obtenir un tri stable, il faut changer la condition « `L[i] <= L[0]` » par « `L[i] < L[0]` ».

Exercice 4. Tri par comptage

Question 1 –

```
# Hypothèse: L est non vide
def maxi(L):
    m = L[0]
    for i in range(len(L)):
        if L[i] > m:
            m = L[i]
    return m
```

```
def comptage(L):
    if L == []:
        return []
    M = [0 for _ in range(maxi(L)+1)]
    for e in L:
        M[e] += 1
    return M
```

```
def tri_comptage(L):
    T = []
    M = comptage(L)
    for i in range(len(M)):
        for _ in range(M[i]):
            T.append(i)
    return T
```

Question 2 –

```
# Hypothèse: L est non vide
def mini_bis(L):
    m = L[0]
    for i in range(len(L)):
        if L[i] < m:
            m = L[i]
    return m
```

```
def comptage_bis(L):
    if L == []:
        return [], []
    M_pos = [0 for _ in range(maxi(L)+1)]
    M_neg = [0 for _ in range(-mini_bis(L)+1)]
    for e in L:
        if e >= 0:
            M_pos[e] += 1
        else:
            M_neg[-e] += 1
    return M_pos, M_neg
```

```
def tri_comptage_bis(L):
    T = []
    M_pos, M_neg = comptage_bis(L)
    for i in range(len(M_neg)-1, -1, -1):
        for _ in range(M_neg[i]):
            T.append(-i)
    for i in range(len(M_pos)):
        for _ in range(M_pos[i]):
            T.append(i)
    return T
```

Exercice 5. Tri de crêpes

Question 1.a –

```
def maxi(L, i):
    j = i
    for k in range(i+1, len(L)):
        if L[k] > L[j]:
            j = k
    return j
```

Question 1.b –

```
def spatule(L, i):
    taille_sous_liste = len(L) - i
    for k in range(taille_sous_liste // 2):
        k1 = i + k
        k2 = len(L) - 1 - k
        tmp = L[k1]
        L[k1] = L[k2]
        L[k2] = tmp
```

Question 2.a – On va faire une sorte de tri par sélection : on place la plus grande crêpe à l'indice 0, puis la deuxième plus grande à l'indice 1, et ainsi de suite. Pour cela, on fait varier un indice $i \in [\![0, n-2]\!]$, puis :

→ On repère l'indice $j \geqslant i$ de la crêpe avec le plus grand diamètre (opération 1).

→ Avec un coup de spatule à l'indice j (opération 2), on place la crêpe sur le dessus de la pile.

→ Avec un deuxième coup de spatule à l'indice i (opération 2), on place la crêpe à l'indice i.

Remarque. Pour diminuer le nombre d'appels à la fonction `spatule`, on pourrait faire des cas spéciaux lorsque $j = n-1$ (un coup de spatule suffit) et lorsque $j = 0$ (les appels à la fonction `spatule` ne sont pas utiles).

Question 2.b – Notons a_n et b_n le nombre maximal d'utilisations des opérations 1 et 2 pour une liste de taille n. On a :

$$a_0 = a_1 = 0, \qquad b_0 = b_1 = 0.$$

Lorsque $n \geqslant 2$, on utilise une fois l'opération 1, puis deux fois l'opération 2, puis on trie une pile de taille $(n-1)$. Pour tout $n \geqslant 2$, on a donc :

$$a_n = a_{n-1} + 1 \qquad b_n = b_{n-1} + 2$$

En résumé :

$$\begin{array}{lll} a_0 = 0, & \text{et} & a_n = n-1 \text{ pour tout } n \geqslant 1 \\ b_0 = 0, & \text{et} & b_n = 2n-2 \text{ pour tout } n \geqslant 1 \end{array}$$

Question 3.a – Notez que la description faite dans la question 2.a correspond à une fonction récursive alors que le code ci-dessus est une fonction itérative. Il faut donc adapter la procédure en conséquence.

```
def tri_crepes(L):
    for i in range(len(L)-1):
        j = maxi(L, i)
        spatule(L, j)
        spatule(L, i)
```

Question 3.b – Les fonctions `maxi` et `spatule` s'exécutent en temps linéaire en `len(L)`. Dans la fonction `tri_crepes`, il y a un nombre linéaire de tours de boucle et chaque tour de boucle s'exécute en temps linéaire. Finalement, la complexité est quadratique en `len(L)`.

Question 4 –

```
def tri_crepes_bis(L):
    T = L[:]
    tri_crepes(T)
    return T
```

Question 5 – On va générer toutes les permutations de la liste `L = [n-1, n-2, ..., 0]` et déterminer le nombre minimal d'appels à la fonction `spatule` pour trier chaque permutation. Pour cela on va procéder à l'envers : on part de `L`, on génère toutes les listes obtenues avec un appel à la fonction `spatule` (il y en a `n`), puis on recommence avec ces nouvelles listes jusqu'à ce que toutes les permutations aient été atteintes.

Le nombre d'appels à la fonction `spatule` nécessaires pour atteindre une permutation `P` sera stocké dans un dictionnaire. Étant donné que les clés d'un dictionnaire ne peuvent pas être des listes, on utilise `tuple(P)` comme clé.

```
def make_dict_nb_etapes(n):
    """
    n: int
    Return: int, dict[tuple, int]
    ----
    Renvoie le nombre maximal d'étapes ainsi qu'un dictionnaire qui associe à
    chaque liste L le nombre d'étapes nécessaires pour obtenir L.
    """
    L = [i for i in range(n, 0, -1)]
    d = {}
    etape = 0
    d[tuple(L)] = etape
    derniers_vus = [L[:]]
    while derniers_vus != []:
        etape += 1
        nouveaux_vus = []
        for M in derniers_vus:
            for i in range(n):
                N = M[:]
                spatule(N, i)
                Nt = tuple(N)
                if Nt not in d:
                    d[Nt] = etape
                    nouveaux_vus.append(N)
        derniers_vus = nouveaux_vus
    return etape-1, d
def nb_max_op2(n):
    n, _ = make_dict_nb_etapes(n)
    return n
```

Exercice 6. Tri à bulles

Question 1 –

```
def etape_tri_bulles(L, i_max):
    for i in range(i_max+1):
        if L[i] > L[i+1]:
            tmp = L[i]
            L[i] = L[i+1]
            L[i+1] = tmp
```

```
def tri_bulles(L):
    for i_max in range(len(L)-2, -1, -1):
        etape_tri_bulles(L, i_max)
```

Question 2 – La fonction `etape_tri_bulles` est en complexité Θ(`i_max`). Donc la fonction `tri_bulles` est en complexité $\Theta(\texttt{len(L)}^2)$.

Question 3 – C'est bien un tri en place, puisque la quantité de mémoire utilisée est indépendante de la taille de la liste en entrée.

C'est un tri stable. Le point important est que dans la fonction `etape_tri_bulles` deux éléments égaux ne sont pas échangés. Ainsi, le tri ne serait pas en stable si on remplaçait la condition « `L[i] > L[i+1]` » par « `L[i] >= L[i+1]` ».

Exercice 7. Le tri du singe

Question 1 –

```
def est_triee(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

```
# Cette fonction renvoie l'indice du a ème None dans S.
# Hypothèse: cet indice existe.
def a_eme_None(a,S):
    nb_None_vus = 0
    i = 0
    while nb_None_vus < a:
        if S[i] is None:     ## Ou (if S[i] == None:)
            nb_None_vus += 1
        i = i+1
    return i-1
```

```
def perm_alea(n):
    S = [None]*n
    for i in range(n):
        A = n-i
        a = random.randint(1,A)
        S[a_eme_None(a,S)] = i
    return S
```

```
def appliquer_perm(L,S):
    M = [None]*len(S)
    for i in range(len(S)):
        M[i] = L[S[i]]
    return M
```

```
def tri_singe(L):
    N = L[:]
    n = len(L)
    while not est_triee(N):
        S = perm_alea(n)
        N = appliquer_perm(N,S)
    return N
```

Question 2 – Ce n'est pas un tri en place car une nouvelle liste est créée.
Ce n'est pas un tri stable, puisque deux éléments égaux peuvent être échangés.